Hierarchical structuring of scenes with MKtrees

Omar Rodríguez González
Facultad de Ingeniería
Universidad Autónoma
de San Luis Potosí
San Luis Potosí, México
omarg@uaslp.mx

Marta Franquesa Niubó
Departament de Llenguatges
i Sistemes Informàtics
Universitat Politècnica de Catalunya
Barcelona, España
marta@lsi.upc.edu

13th January 2005

Abstract

The location of an object or parts of an object in huge environments is of fundamental importance in several research areas. The problem to locate an object or parts of an object in very complex systems (environments with large number of objects) has been studied by several authors, contributing with a large number of solutions based on different hierarchical representations of the scenes.

In this research report, an implementation of a Multiresolution K dimensional tree (Multiresolution Kdtree, MKtree) and its results are presented. A MKtree represents a hierarchical subdivision of the scene objects, that guarantees a minimum space overlap between node regions in large environments. The implementation is related to ship design applications where the number and distribution of objects are considered complex. This will serve for later inclusion of methods to better approximate objects with tighter bounding volumes.

1 Introduction

The realism needed in several areas such as robotics, computer animation and any application where large environments (also called complex systems) are used has stimulated the need to research for efficient techniques in model simplification, mesh compression, real-time rendering, and collision detection tests.

Referring to the collision detection problem, the term hybrid collision detection, introduced by Kitamura et al. in [KTAK94], arises when more than two objects are moving in the same space. The hybrid collision detection refers to any collision detection method which first performs one or more iterations of approximate test to study whether objects interfere in the workspace and

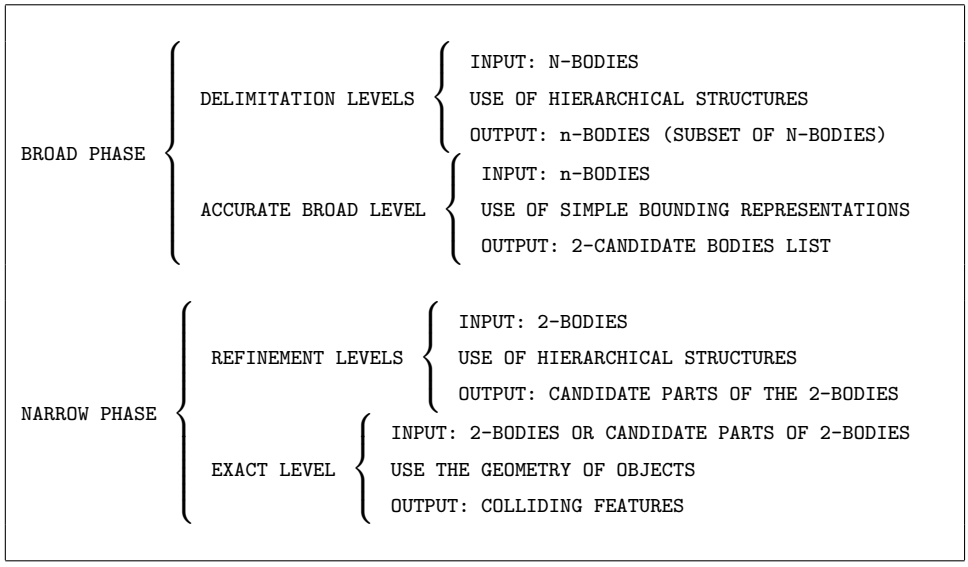

Figure 1: Phases of the global collision detection problem

then, performs a more accurate tests to identify the objects parts causing the interference.

Hubbard [Hub95] reports two collision tests encompassing two phases: the *broad phase*, where approximate interferences are detected, and the *narrow phase* where exact collision detection is performed. O'Sullivan and Dingliana [O'S99, OD99] extended the hybrid collision detection classification phases pointing out that the *narrow phase* itself consists of several levels of intersection testing between two objects at increasing level of accuracy, where the last one may be exact. They referred the most accurate level as *narrow phase: exact level* and the phase of testing for collision using increasing levels of accuracy as *narrow phase: progressive refinement levels*. Franquesa and Brunet [FNB03] extended the *broad phase* pointing that it may consist of two subphases too. In the first one, tests are performed to find subsets of objects of the entire workspace where collisions can occur, rejecting, at the same time, all the space regions where interference is not possible. In the second subphase, the collision test determines the candidate objects that can cause collision. They refer the first subphase as *broad phase: progressive delimitation levels* and, the second subphase as *broad phase: accurate broad level*. Figure 1 summarizes the main features of each phase.

The *broad phase* is well suited not only for the collision detection problem, but also can be applied to other techniques as, real-time rendering, view-dependent rendering, etc.

This research report is based on the work of Franquesa in [FN04], in which the *broad phase* is covered with the proposal of the MKtrees, a data structure

useful when performing simultaneously space and scene subdivision. The *narrow phase* is approached using a collision prediction solution based on the work of Kim and Rossignac [KR03], in which the time and location of collisions are computed directly from the relative motion of pairs of objects.

A software system for MKtrees construction is implemented and is presented as the foundation for future research (see section 5). One of the main methods for MKtrees construction and its results are evaluated and visual output results are obtained from several samples.

This software implementation creates a complete MKtree structure of a complex scene (in this case an oil tanker), and writes a new file with the results of the first subdivision in the hierarchical structure, whose visualization gives an overall idea about the results of the algorithms.

1.1 Previous related work

We give a brief overview on the related work applied to different application contexts.

1.1.1 Spatial partition hierarchies

This methods decomposes the space in a hierarchical way. The Regular Grids of Boxes model is based on a grid of equal sizes boxes over the space (also called *voxels*), which can be unit cubes [HKM95, MPT99], or parallelepipeds [GASF94]. Thus, objects are presented by a 3D matrix of voxels. The octrees [Sam90] is a tree of degree eight (octants) which represents the space occupied by objects contained in a space defined by cubes. But not only cubes are used in octrees, [Hub93, PG95] used spheres trees. A Bintree [ST85] is a recursive dimension-independent subdivision of the space, whose cuts are exactly half of the original space and orthogonal to the coordinate axis. A Binary Space Partition tree (BSPtree) [TN87] recursively divides the space into two subspaces, each separated by a plane of arbitrary orientation and position. A Kdtree [Ben75], is a binary space partition tree whose cuts are orthogonal to the coordinate axis. A Multiresolution Kdtree (or MKtree) [FNB03] represents a hierarchical subdivision of a scene, that guarantees a minimum space overlap between node regions in large environments.

1.1.2 Bounding volume hierarchies

The bounding volume hierarchies approximate a representation of an object as representation hierarchy, known as Bounding Volume tree (BVtree). The SphereTrees [Hub93] represents objects by sets of spheres. Two methods are commonly used for the construction of the SphereTree, *medial-axis* surface [Qui94, Hub95, Hub96, OD99, BO03], and fitting spheres to a polyhedron

and shrinking them until they just fit [RB79]. The Axis-Aligned Bounding Box trees (AABBtrees) [BKSS90] are based in boxes with axis-aligned faces. An Oriented Bounding Box tree (OBBtree) [GLM96] represents an object by an enclosed oriented box. The kDOP trees [KHM+98] are a set of discrete oriented polytopes bounding the volume. In [YSLM04], a clustered hierarchy of progressive meshes is used as a dual hierarchy, a BV hierarchy with clusters bounded by OBB, and a multiresolution hierarchy of the object. [LAM03] proposed a bounding-volume tree over morphing objects for collision detection, updating the tree with the same morphing function used for the vertices of the object. [JP04] create a Bounded Deformation Tree (BD-Tree) with spheres as BV for fast collision detection with generic deformable models.

1.2 The problem to be solved

Create a suitable framework to continue testing and researching with MKtrees is the primary goal of this research report. This will allow us to explore new methods applied over objects in complex systems. The problem is directed to rigid polyhedra in fixed localizations in space.

The main interest is related to create a hierarchical structure of a complex environment in ship design applications that will serve to incorporate new approximative representations of scenes and objects as sphere-trees or kDOP trees. This structure, combined with the approximative representations, will help us to speed up different tests (point classification, collision detection, ...).

This implementation develops one of the three MKtrees construction algorithms presented in the section 4.2 of [FN04], the Splitting Overlap Algorithm (SOA). This algorithm distributes objects of the scene minimizing the number of objects that intersect the overlapping space between node regions, and creates another structure (*SplitTree*) inside the overlapping space.

2 MKtrees structure

This class of tree represents a hierarchical subdivision of the scene objects that guarantees a minimum space overlap between subtree regions. This structure is useful for collision detection queries and objects classification.

2.1 Description

The MKtrees has the following features:

- The MKtree represents a hierarchical subdivision of the objects in the scene that minimizes the number of objects in the overlap regions.

- The algorithms for MKtree simultaneously partition the space and the scene objects.
- Bounding approximations of objects can be used.
- The method has been conceived to manage memory efficiently.

The MKtrees are useful to treat objects inside a ship design environment, where most objects are pipes (mainly, 1D), walls (mainly, 2D) and some equip-elements (3D). Objects usually are oriented to coordinate axis, as pipes, or oriented to coordinate planes, as walls. The objects in the scene are rigid and static solids.

Before describing the MKtrees, some related definitions are presented:

Definition 2.1.1 *The environment, S, is defined as a collection of 3D objects, usually polyhedra. Thus,*

$$S = \{o_1, \ldots, o_N\}$$

Definition 2.1.2 *$R(S)$ is the axis-aligned bounding box, AABB, of the set S. $R(S_n)$ is the AABB of a subset S_n, with $S_n \subset S$*

Assumption: There exists a function $InPage(S_n)$ that returns true if all the geometry of S_n fits in the space reserved for one tree leaf - d disk blocks (and can be retrieved to the core memory in d disk access operations):

function $InPage(S_n)$ **return bool**
  return $(GeometrySize(S_n) \leq MemPage)$
end function

Where $MemPage$ is a global constant that indicates the size of one leaf. $MemPage = d * OneBlockSize$

We can describe a MKtree as follows:

A MKtree is a tree, $MKT(S)$, that specifies a hierarchy on S, where S is a set of objects. Each node, n, of the $MKT(S)$ corresponds to a subset $S_n \subset S$, where the root is associated with the full set S. Each internal node n has two children. The union of the object subsets associated to the children of n is equal to S_n. Each node n is associated to the AABB box of $S_n : R(S_n)$. S_{n_1} and S_{n_2} are the subsets associated with the children of n. Assume that exists a $wDivideList$ procedure that divides the collection of objects belonging to S_n in two subsets S_{n_1}, and S_{n_2} minimizing the overlap between $R(S_{n_1})$ and $R(S_{n_2})$ (where w can be x, y or z).

The performance of a MKtree is related to the selection of the splitting rules to build the hierarchy. The main goal is that during the tree construction

subsets S_{n_1} and S_{n_2} of objects can be assigned to each child of a node, n, in such a way that they can be spatially separated.

A node of a MKtree can be defined as:

- If $InPage(S_n)$, then n is a leaf node and stores the *geometry* of objects in S_n
- If *no* $InPage(S_n)$, then n is a leaf node with information on the overlap region and two pointers to n_1 and n_2, respectively. S_{n_1} and S_{n_2} are obtained by the $wDivideList$ minimum overlap value where w can be x, y or z.

Given a node n, the procedure $wDivideList$ splits the set of objects S_n into subsets S_{n_1} and S_{n_2} taking into account the direction in which the minimum overlap has been found, $SplitDir$. Thus the $SplitDir$ is a unit vector being (1,0,0), (0,1,0) or (0,0,1).

Definition 2.1.3 *π_0 and π_1 are two oriented half-spaces that bound the overlap region:*

- *π_0 is the half-space of the first face of $R(S_{n_2})$ in the SplitDir direction*
- *π_1 is the half-space of the last face of $R(S_{n_1})$ in the SplitDir direction*
- *Every object of $R(S_{n_2})$ is in π_1*
- *Every object of $R(S_{n_1})$ is in π_0*
- *The overlap region is the intersection of $R(S_n)$, π_0 and π_1*
- *The overlap size is the distance between the planes π_0 and π_1*

In this way the tree construction procedure generates a subdivision and a hierarchy of all objects of S with a minimum number of objects intersecting the overlap space. The method to implement performs a spatial splitting of the overlap region using a compact KdTree, the *SplitTree* (*SpTree*).

Each node n of the MKtree, corresponding to a subset S_n, is defined as:

- If $InPage(S_n)$, then n is a leaf node and stores a pointer to the *geometry* of objects in S_n
- If *no* $InPage(S_n)$, then n is an internal node and it is represented by:
  - The splitting direction $SplitDir$
  - The bounding box $R(S_n)$
  - The planes defining the overlap region: π_0 and π_1

- Two pointers to the son nodes n_1 and n_2 (associated to the object sets S_{n_1} and S_{n_2}
- The corresponding $SpTree$ (hierarchical set of half spaces).

The internal nodes of the MKtree (including the SpTrees of the nodes) are stored in main memory. Leaf nodes storing the geometry of the objects use out-of-core storage.

3 MKtrees construction

The Splitting Overlap Algorithm (SOA) distributes objects over the children minimizing the number of objects that intersect the overlapping space between node regions. Then, the algorithm computes an *SpTree* with the objects that belong to the overlapping spaces.

The *wDivideList* for the SOA algorithm is based on the following concepts:

1. Objects that belong to S_n are sorted in increasing order of w coordinate ($w = x$, y or z).

2. The w maximum coordinate value of objects that belong to S_{n_1} ($S_{n_1} = o_1..o_{ft}$, ft = first to be treat) of the objects set S_n, determines the half-space π_0.

3. The w minimum coordinate value of objects that belong to S_{n_2} ($S_{n_2} = o_{N-lt}..o_N$, lt = last to be treat and N = number of objects in S_n) of the objects set S_n, determines the half-space π_1.

4. The space between π_0 and π_1 is the overlapping space between the regions of the subtrees.

5. Objects in S_n are classified in several groups depending on their positions with respect to half-spaces π_0 and π_1, as follows:

   - $WWObjects$: Group of objects that fall on the left of π_0. In other words, $WWObjects$ = Set of objects o_i such that $o_i \in S_{n_1}$ and $Classif(o_i, \pi_0) = out$. This group of objects will be included to the subset S_{n_1}.
   - $BBObjects$: Group of objects that fall on the right of π_1. In other words, $BBObjects$ = Set of objects o_i such that $o_i \in S_{n_2}$ and $Classif(o_i, \pi_1) = out$. This group of objects will be included to the subset S_{n_2}.
   - $OverObjects$: Subset of objects, say S_m, that intersects the over-space limited by π_0 and π_1. This group of objects will be used to compute the SpTree. Then, S_m will be distributed over S_{n_1} and

S_{n_2}, before building the subtree for S_n. S_m is made out of three subsets S_w, S_B and S_{In} such that:

- S_{In} (or $InObjects$): Objects that fall between π_0 and π_1, this means that are totally inside the overspace. In other words, S_{In} = Set of objects o_i such that $Classif(o_i, \pi_0) = in$ and $Classif(o_i, \pi_1) = in$.
- S_W (or $WObjects$): Objects that only intersects the π_0 plane. In other words, S_W = Set of objects o_i such that $o_i \in S_{n_1}$ and o_i is neither a $WWObject$ nor an $InObject$.
- S_B (or $BObjects$): Objects that only intersects the π_1 plane. In other words, S_B = Set of objects o_i such that $o_i \in S_{n_2}$ and o_i is neither a $BBObject$ nor an $InObject$.

$WObjects$ are intersected by the plane of π_0, whereas $BObjects$ are intersected by the plane of the the half-space π_1. Note that the way π_0 and π_1 planes are selected, does not allow objects to intersect the two planes at the same time.

3.1 Algorithms description

The construction of the software was made using the P3D library used in the Alice [1] software to read the p3d file containing the oil tanker of the complex scene. While reading the geometric information from the file, an $AABB$ for each element in the scene is constructed and stored in main memory. Better bounding volumes with the MKtrees can be used, instead of $AABB$, but this remains as future research (see section 5).

With the $AABB$ of each object (or *level of detail* = 1, $LOD = 1$), the elements are processed to build the MKtree by the SOA algorithm. Colors are assigned to the elements after the first subdivision occurs in the algorithm. At the end, the $AABB$ representation of each object in the MKtree is written with the P3D library into an output p3d file which can be visualized using the Alice software.

The procedure $BuildMKtree$ is the implementation of the SOA algorithm (see algorithm 1). It has six arguments, the actual MKtree node mkt, the list of objects S_n starting from $geom_{ini}$ to $geom_{end}$, the number of objects N, the limit, $ratio$, of the sublist of objects that will be examined r (with $0 \leq r \leq 1$), and the $level$ tree in which the nodes will be colorized. The parameter r helps the tree to be more or less balanced depending on its value. When splitting a set of objects S_n with N objects, $S_n = \{o_1, \ldots, o_N\}$, the sublist of objects to be examined is defined by the following rank of object indices: $\{o_k\}$ with $k \in [r * N..(1.0 - r) * N]$.

[1] http://www.lsi.upc.es/dept/crv/webcrv/investigacion_ing_archivos/alice.htm

procedure $BuildMKtree(mkt, geom_{ini}, geom_{end}, N, r, level)$
  if no $InPage(geom_{ini}, geom_{end})$ **then**
    {Initialize mkt}
    $fT = First(geom_{ini}, N, r)$
    $fT = Last(geom_{end}, N, r)$
    {ft = firstTreat, lT = lastTreat}
    for $SpDir$ **in** $[x, y, z]$ **do**
      $SortGeom(geom_{ini}, N, SpDir, min)$
      $MinObjOver(geom_{ini}, geom_{end}, fT, lT, SpDir, min, minOb)$
      $SortGeom(geom_{ini}, N, SpDir, max)$
      $MinObjOver(geom_{ini}, geom_{end}, fT, lT, SpDir, max, minOb)$
    end for
    $SubSetOverlapObj(minOb, geom_{ini}, S_{n_1}, S_{n_2}, S_m)$
    $Sp = gnSplitTree(S_m, mkt)$
    if $level == 1$ **then**
      $AssignColors(S_m, S_{n_1}, S_{n_2})$
    end if
    $DivideSets(mkt, S_m, S_{n_1}, S_{n_2})$
    $BuildMKtree(mkt_{leftchild}, S_{n_1 ini}, S_{n_1 end}, N, r, (level - 1))$
    $BuildMKtree(mkt_{rightchild}, S_{n_2 ini}, S_{n_2 end}, N, r, (level - 1))$
  else
    {MkTree node is a terminal node and points to geometry}
    return
  end if
end procedure

Algorithm 1: Build a MKtree

The algorithm 1 works as follows: when the total size of the geometry of S_n is bigger than the user defined block size ($InPage(geom_{ini}, geom_{end})$ is $false$), the best dimension and location is computed for $SpDir$, sorting six times the actual objects list ($SortGeom$ procedure) and finding the minimum object overlap region ($MinObjOver$ procedure). The resulting region is stored in the structure $minOb$. The actual list of objects is then distributed over three subsets: S_{n_1}, S_{n_2} and S_m, taking into account the half-spaces π_0 and π_1 ($SubSetOverlapObj$ procedure). The third subset, S_m, corresponds to the minimum set of objects that intersect the overlapping space, and used to compute and create a SpTree Sp ($gnSplitTree$ function). If the MKtree level is 1, colors are assigned to all the elements in the three subsets S_{n_1}, S_{n_2} and S_m ($AssignColors$ procedure). Objects in S_m are then distributed in the subsets S_{n_1} and S_{n_2}, which are recursively called in the $BuildMKtree$ procedure.

procedure $MinObjOver(geom_{ini}, geom_{end}, fT, lT, SpDir, min, minOb)$
  $\pi_0 = fT$
  $\pi_1 = lT$
  for $\forall\ o_i \in S_n$ **do**
    if intersects o_i with π_0 and π_1 **then**
      {Fix π_0 or π_1 so the object intersects only one plane}
    end if
  end for
  {Store number of objects intersecting, $SpDir$, π_0 and π_1 in $minOb$}
end procedure

Algorithm 2: Minimum object overlap

The algorithm 2 detects if an object in S_n intersects both π_0 and π_1. If this is the case, π_0 or π_1 are modified and the algorithm test again all the objects of S_n until no object intersects with both planes. When no objects in S_n intersects π_0 and π_1, the structure $minOb$ is filled with the information required.

function $gnSplitTree(S_m, mkt)$ **return** $SplitTree$
  $s : SplitTree$
  if $intersection(mkt_{boxregion}, S_m)$ **then**
    $FindBestSplittingOrientedPlane(S_m, mkt, spl, spl_{comp})$
    $s_{halfspace} = spl$
    $s_{leftchild} = gnSplitTree(S_m, spl)$
    $s_{rightchild} = gnSplitTree(S_m, spl_{comp})$
    return s
  end if
  return $null$
end function

Algorithm 3: Generate a SplitTree

The algorithm 3 is a recursive function that computes an $SpTree$ with the objects in the S_m set. It recursively computes the best dimension and location to divide the initial list to minimize the number of objects that overlap the intersection space. In fact, the $SplitTree$ is a kind of $Kdtree$ that splits the overlap region.

The procedure $FindBestSplittingOrientedPlane$ restricts itself to isothetic planes in $mkt_{boxregion}$. It works by trying to find a plane either with the

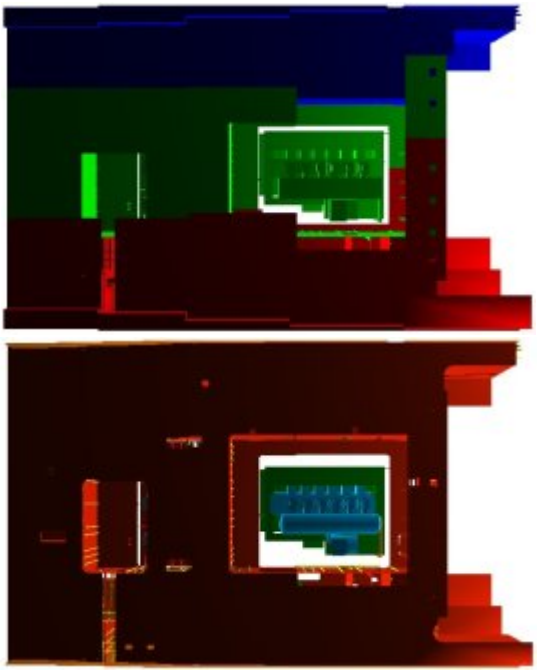

Figure 2: SC33 oil tanker. Top: scene with MKtree level 1 and LOD = 1. Bottom: original scene

minimum intersection with S_W, or with the minimum intersection with S_B, and returns an oriented plane (half-space).

For the source code of the above algorithms, see appendix A.

4 Results and discussion

The p3d resulting file, can be visualized with the Alice program and is a graphic representation of a constructed MKtree. This is only an indicative reference of what the algorithm can achieve. The complete results of each node of the MKtree can not be visualized, except in data form inside the compiler debug mode.

The figure 2 and figure 3 were processed with input argument values $r = 0.3$ and $MemPage = 12kb$. The top image represents an oil tanker visualized from a top-view perspective with $LOD = 1$ and, in blue color and red color, objects that belong to different spatial disjoint spaces ($WWObjects$ and $BBObjects$ respectively), and in green, objects in the overlap region processed with a SpTree ($OverObjects$). The bottom image corresponds to the original scene with $LOD = max$.

For more examples, see figure 5 and figure 6 in appendix B.

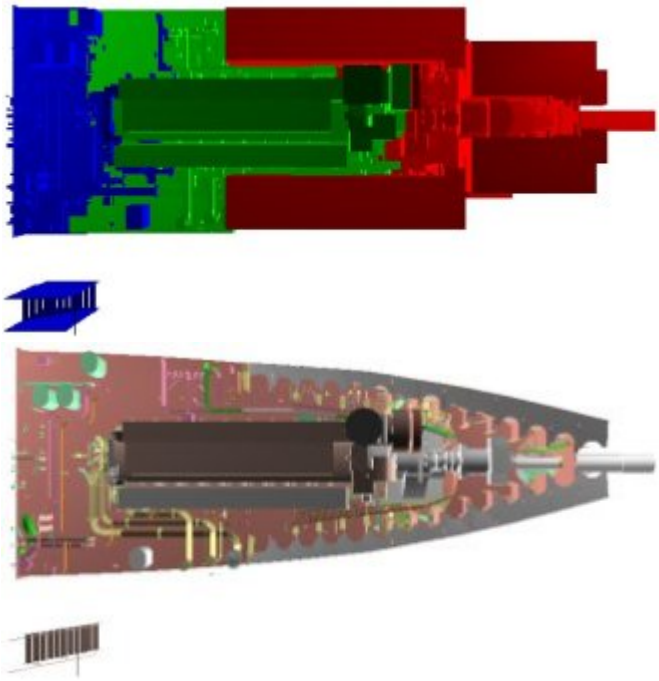

Figure 3: Astano oil tanker. Top: scene with MKtree level 1 and LOD = 1. Bottom: original scene.

4.1 Usage of the software

The system that generates the MKtrees, has an easy to use interface (figure 4). Actually, the system doesn't present any visual result at run-time, other than the dialogs of success or error in the construction of the MKtree and in the writing success of the resulting p3d file. Because of this, the main system frame contains only one menu element. Usage is as follows:

OpenP3D : Opens a p3d file containing a scene with an oil tanker. At the moment of the opening, the software reads all the objects in the scene and constructs a MKtree for the scene. The successful construction of the MKtree is indicated by two dialogs, one for the correct reading of the p3d file geometry, and another one for the correct construction of the MKtree in memory. Depending on the size of the scene and hardware specifications, this process can take some time.

WriteP3D : Writes a new p3d file containing the $AABB$ of the objects from a scene previously loaded. The LOD will be equal to 1 and the colors assigned will represent the result of the first subdivision (level 1 of the MKtree) of the SOA algorithm. If an MKtree is not in memory (constructed first via OpenP3D), a dialog will inform about the error.

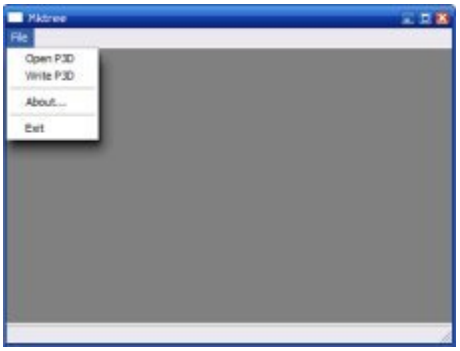

Figure 4: Main frame of the MKtree generation software

About : Shows a message dialog with information about the system.

Exit : Exits the software.

4.2 Technical details

The implementation was made using dynamic allocated memory, so the number of objects in the scene was not a concern. The software was created using the Dev-C++ [2] compiler, under a Windows XP platform, and using wxWindows [3] as API for the GUI interface. The system could be ported easily to different platforms, but this was not tested. It was used C++ for the GUI development, and C for the implementation of the rest of algorithms. The amount of RAM needed, depends directly on the complexity of the scene.

5 Conclusions and future work

In this research report, a system for creating MKtrees has been presented, with a complex environment as its input, and a new output file which represents the visual result of the construction.

The motivation of this work was to create a suitable framework to continue testing and researching with MKtrees as the foundation to introduce new data structures, as sphere trees or kDOP trees for better approximation of the objects in the scene. This will permit us to explore new accurate methods to accelerate different tests in complex systems.

Actually, the uses and improvements over MKtrees provide a vast area to research in both phases, *broad phase* and *narrow phase*, of the collision detection problem and other research areas where complex systems are used.

[2] http://www.bloodshed.net/devcpp.html
[3] http://www.wxwindows.org

Acknowledgements

This research has been partially supported by the Ministerio de Ciencia y Tecnología under the project MAT2002-0497-C03-02 and the Facultad de Ingeniería of the Universidad Autónoma de San Luis Potosí under the PROMEP program.

References

[Ben75] J.L. Bentley. Multidimensinal binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 322–331. ACM Press, 1990.

[BO03] G. Bradshaw and C. O'Sullivan. Adaptative Medial–Axis Approximation for Sphere–Tree Construction. *ACM Transactions on Graphics*, 22(4), 2003.

[FN04] M. Franquesa-Niubò. *Collision Detection in Large Environments using Multiresolution KdTrees*. PhD thesis, Universitat Politècnica de Catalunya, March 2004.

[FNB03] M. Franquesa-Niubo and P. Brunet. Collision detection using *MKtrees*. In *Proc. CEIG 2003*, pages 217–232, July 2003.

[GASF94] A. Garcia-Alonso, N. Serrano, and J. Flaquer. Solving the collision detection problem. *IEEE Computer Graphics and Applications*, pages 36–43, May 1994.

[GLM96] S. Gottschalk, M.C. Lin, and D. Manocha. *OBBtree*: A hierarchical structure for rapid interference detection. In *ACM SIGGRAPH Conf. Proc.*, pages 171–180, August 1996.

[HKM95] M. Held, J.T. Klosowski, and J.S.B. Mitchel. Evaluation of collision detection methods for real reality fly–throughs. In C. Gold and J.M. Robert, editors, *Proc. seventh Canadian Conf. on Computer Geometry*, pages 205–210, August 1995.

[Hub93] P. M. Hubbard. Interactive collision detection. In *Proc. IEEE Symp. on Research Frontiers in Virtual Reality*, volume 1, pages 24–31, October 1993.

[Hub95] Philip M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, September 1995.

[Hub96] Philip M. Hubbard. Aproximating polyhedra with spheres for time–critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, July 1996.

[JP04] Doug L. James and Dinesh K. Pai. BD-Tree: Output-sensitive collision detection for reduced deformable models. *ACM Transactions on Graphics (SIGGRAPH 2004)*, 23(3), August 2004.

[KHM+98] J. T. Klosowski, M. Held, J. S.B. Mitchel, H. Sowizral, and K. Zikan. Eficient collision detection using bounding volume hierarchies of k–dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, january-march 1998.

[KR03] B. Kim and J. Rossignac. Collision prediction for polyhedra under screw motions. In *Proc. of the ACM SM'03*, pages 4–10, June 2003. Seatle, Washington.

[KTAK94] Y. Kitamura, H. Takemura, N. Ahuja, and F. Kishino. Efficient collision detection among objects in arbitrary motion using multiple shape representation. In *Proceedings 12th IARP Inter. Conference on Pattern Recognition*, pages 390–396, October 1994.

[LAM03] T. Larsson and T. Akenini-Mller. Efficient collision detection for models deformed by morphing. *The Visual Computer*, 19(2-3):164–174, May 2003.

[MPT99] W.A. McNeely, K.D. Puterbaugh, and J.J. Troy. Six–degrees–of–fredom haptic rendering using voxel sampling. In *Proceedings of SIGGRAPH 99*, pages 401–408, August 1999. ISBN: 0–20148–560–5.

[OD99] C. O'Sullivan and J. Dingliana. Real–time collision detection and response using sphere–trees. In 15th Spring Conference on Computer Graphics, April 1999. ISBN: 80–223–1357–2.

[O'S99] C. O'Sullivan. *Perceptually–Adaptive Collision Detection for Real–time Computer Animation.* PhD thesis, University of Dublin, Trinity College Department of Computer Science, June 1999.

[PG95] I.J. Palmer and R.L. Grimsdale. Collision detection for animation using sphere-trees. *Computer Graphics Forum*, 1995.

[Qui94] S. Quinlan. Efficient distance computation between non–convex objects. In *Proceedings of the IEEE Int. Conf. on Robotics and Automation*, pages 3324–3329, 1994. San Diego, CA.

[RB79] J.O. Rourke and N. Badler. Decomposition of three–dimensional objects into spheres. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI–1(3):295–305, July 1979.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990. ISBN 0-201-50255-0.

[ST85] H. Samet and M. Tamminen. Bintrees, *CSG* trees, and time. In *ACM SIGGRAPH Conf. Proc.*, pages 121–130, July 1985. Computer Graphics vol. 19 num. 3.

[TN87] W.C. Thibault and B.F. Naylor. Set operations on polyhedra using binary space partitioning trees. In *ACM Computer Graphics*, volume 21, pages 153–162, July 1987.

[YSLM04] Sung-Eui Yoon, Brian Salomon, Ming Lin, and Dinesh Manocha. Fast Collision Detection between Massive Models using Dynamic Simplification. In *Proceedings of the Eurographics Symposium on Geometry Processing*, 2004.

Appendix

A Source code

```
// Splitting Overlap Algorithm (SOA)
void BuildMKTree(mktree *mkt, AABB *inigeom, AABB *endgeom,
UInt32 nobjs, Float32 r, UInt8 level)
{
  splittree *Sp;
  AABB *Sn1 = NULL, *Sn2 = NULL, *Sm = NULL;;
  UInt8 SpDir;
  tminOb minOb;

  if (!InPage(inigeom, endgeom))
  {
    mkt->iniobjects = NULL;
    mkt->endobjects = NULL;

    AABB *newbox = (AABB*) malloc(sizeof(AABB));
    newbox->leftAABB = newbox->rightAABB = NULL;
    newbox->xmin = newbox->ymin = newbox->zmin = 999999.99;
    newbox->xmax = newbox->ymax = newbox->zmax = -999999.99;
    newbox->numvertices = 0;
    AABB *run = inigeom;
    while (run != NULL)
    {
      if (run->xmin < newbox->xmin) newbox->xmin = run->xmin;
      if (run->ymin < newbox->ymin) newbox->ymin = run->ymin;
      if (run->zmin < newbox->zmin) newbox->zmin = run->zmin;
      if (run->xmax > newbox->xmax) newbox->xmax = run->xmax;
      if (run->ymax > newbox->ymax) newbox->ymax = run->ymax;
      if (run->zmax > newbox->zmax) newbox->zmax = run->zmax;
      run = run->rightAABB;
    }
    mkt->boundbox = newbox;

    for (SpDir = XDIR; SpDir <= ZDIR; SpDir++)
    {
      SortGeom(inigeom, 1, nobjs, SpDir, MIN);
      MinObjOver(inigeom, endgeom, First(inigeom, nobjs, r),
       Last(endgeom, nobjs, r), SpDir, MIN, &minOb);
      SortGeom(inigeom, 1, nobjs, SpDir, MAX);
      MinObjOver(inigeom, endgeom, First(inigeom, nobjs, r),
       Last(endgeom, nobjs, r), SpDir, MAX, &minOb);
    }
    SubSetOverlapObj(&minOb, &inigeom, &Sn1, &Sn2, &Sm);

    mkt->overobjects = minOb.nobjs;
    mkt->pi0 = minOb.pi0;
```

```
  mkt->pi1 = minOb.pi1;
  mkt->SpDir = minOb.SpDir;

  Sp = gnSplitTree(Sm, mkt);

  mkt->sptree = Sp;

  if (level == 1)
    AssignColors(Sm, Sn1, Sn2);
  DivideSets(mkt, &Sm, &Sn1, &Sn2);

  mktree *leftmkt = (mktree*) malloc(sizeof(mktree));
  mkt->leftchild = leftmkt;
  mktree *rightmkt = (mktree*) malloc(sizeof(mktree));
  mkt->rightchild = rightmkt;

  run = Sn1;
  UInt32 nobjs = 0;
  UInt8 band = 0;
  while (run != NULL && !band)
  {
    nobjs++;
    if (run->rightAABB != NULL)
      run = run->rightAABB;
    else
      band = 1;
  }
  BuildMKTree(leftmkt, Sn1, run , nobjs , r, (level - 1));

  run = Sn2;
  nobjs = 0;
  band = 0;
  while (run != NULL && !band)
  {
    nobjs++;
    if (run->rightAABB != NULL)
      run = run->rightAABB;
    else
      band = 1;
  }
  BuildMKTree(rightmkt, Sn2, run , nobjs , r, (level - 1));
}
else
{
  // mktree points to geometry nodes
  mkt->iniobjects = inigeom;
  mkt->endobjects = endgeom;
  mkt->overobjects = 0;
  mkt->pi0 = 0;
```

```
    mkt->pi1 = 0;
    mkt->SpDir = -1;
    mkt->sptree = NULL;
    mkt->leftchild = NULL;
    mkt->rightchild = NULL;
  }
}

void MinObjOver(AABB *inigeom, AABB *endgeom, AABB *ft,
AABB *lt, UInt8 SpDir, UInt8 order, tminOb *minOb)
{
  AABB *run = inigeom;
  Float32 pi0, pi1;
  UInt32 nobjs;

  pi0 = retval(ft, SpDir, MIN);
  pi1 = retval(lt, SpDir, MAX);

  run = inigeom;
  UInt8 alt = 0;
  while (run != NULL)
  {
    if (retval(run, SpDir, MIN) < pi0)
      if (retval(run, SpDir, MAX) > pi1)
      {
        if (alt == 0)
        {
          ft = ft->leftAABB;
          pi0 = retval(ft, SpDir, MIN);
          alt = 1;
        }
        if (alt == 1)
        {
          lt = lt->rightAABB;
          pi1 = retval(lt, SpDir, MAX);
          alt = 0;
        }
        run = inigeom;
      }
      else
        run = run->rightAABB;
    else
      run = run->rightAABB;
  }

  nobjs = 1;
  run = ft;
  while (run != lt)
  {
    nobjs++;
```

```
    run = run->rightAABB;
  }
  minOb->overobjs[SpDir][order] = nobjs;

  run = inigeom;
  nobjs = 0;
  while (run != ft)
  {
    nobjs++;
    run = run->rightAABB;
  }
  minOb->ft[SpDir][order] = nobjs;
  while (run != lt)
  {
    nobjs++;
    run = run->rightAABB;
  }
  minOb->lt[SpDir][order] = nobjs;
}

splittree *gnSplitTree(AABB *Sm, mktree *mkt)
{
  splittree *s;
  mktree *spl, *spl_comp;

  if (intersection(mkt, Sm))
  {
    s = (splittree*) malloc(sizeof(splittree));
    spl = (mktree*) malloc(sizeof(mktree));
    spl_comp = (mktree*) malloc(sizeof(mktree));

    FindBestSplittingOrientedPlane(Sm, mkt, &spl, &spl_comp);

    s->spdir = spl->SpDir;
    s->splitval = spl->pi0;
    s->leftchild = gnSplitTree(Sm, spl);
    s->rightchild = gnSplitTree(Sm, spl_comp);

    free(spl->boundbox);
    free(spl);
    free(spl_comp->boundbox);
    free(spl_comp);

    return s;
  }
  return NULL;
}
```

B Further examples

Figure 5: Ship detail. Top left: scene with $r = 0.3$, $MemPage = 12kb$, MKtree level 1 and LOD = 1. Top right: scene with $r = 0.2$, $MemPage = 4.8kb$, MKtree level 1 and LOD = 1. Bottom: original scene

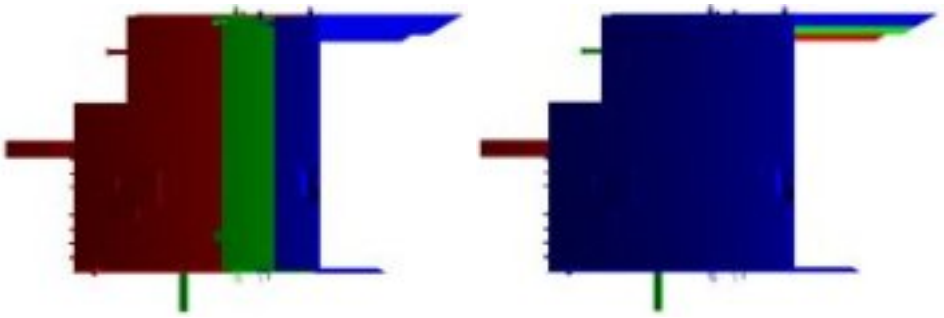

Figure 6: Texaco oil tanker. Left: scene with $r = 0.3$, $MemPage = 12kb$, MKtree level 1 and LOD = 1. Right: scene with $r = 0.2$, $MemPage = 4.8kb$, MKtree level 1 and LOD = 1.